

# Automatic Differentiation via Source Transformation

Jean Utke<sup>1</sup>

<sup>1</sup>University of Chicago and Argonne National Laboratory

SIAM CSE - Minisymposium on Automatic Differentiation  
March 4, 2009



**Argonne**  
NATIONAL  
LABORATORY



UChicago ▶  
Argonne<sub>LLC</sub>



# outline

motivation

basics & examples

- simple forward

- forward with OpenAD

- sample applications

- simple reverse

- preaccumulation & propagation

- reverse with OpenAD

tools and user concerns

- tools

- match tool and application

- source transformation vs. operator overloading

- forward vs. reverse

- checkpointing

summary

# why automatic differentiation?

**given:** some numerical model  $\mathbf{y} = f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$  implemented as a (large / volatile) program

**wanted:** sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

- ➊ don't pretend we know nothing about the program  
(and take finite differences of an oracle)
  - ➋ get machine precision derivatives as  $J\dot{\mathbf{x}}$  or  $\bar{\mathbf{y}}^T J$  or ...  
(avoid approximation-versus-roundoff problem)
  - ➌ the reverse (aka adjoint) mode yields “cheap” gradients
  - ➍ if the program is large, so is the adjoint program, and  
so is the effort to do it manually ... easy to get wrong but hard to debug
- ⇒ use tools to do it **automatically!**

# why automatic differentiation?

**given:** some numerical model  $\mathbf{y} = f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$  implemented as a (large / volatile) program

**wanted:** sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

- ➊ don't pretend we know nothing about the program  
(and take finite differences of an oracle)
  - ➋ get machine precision derivatives as  $J\dot{\mathbf{x}}$  or  $\bar{\mathbf{y}}^T J$  or ...  
(avoid approximation-versus-roundoff problem)
  - ➌ the reverse (aka adjoint) mode yields “cheap” gradients
  - ➍ if the program is large, so is the adjoint program, and  
so is the effort to do it manually ... easy to get wrong but hard to debug
- ⇒ use tools to do it **automatically?**

# why automatic differentiation?

given: some numerical model  $\mathbf{y} = f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$  implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

- ➊ don't pretend we know nothing about the program  
(and take finite differences of an oracle)
- ➋ get machine precision derivatives as  $J\dot{\mathbf{x}}$  or  $\bar{\mathbf{y}}^T J$  or ...  
(avoid approximation-versus-roundoff problem)
- ➌ the reverse (aka adjoint) mode yields “cheap” gradients
- ➍ if the program is large, so is the adjoint program, and  
so is the effort to do it manually ... easy to get wrong but hard to debug

⇒ use tools to do it at least semi-automatically!



# outline

motivation

## basics & examples

simple forward

forward with OpenAD

sample applications

simple reverse

preaccumulation & propagation

reverse with OpenAD

tools and user concerns

tools

match tool and application

source transformation vs. operator overloading

forward vs. reverse

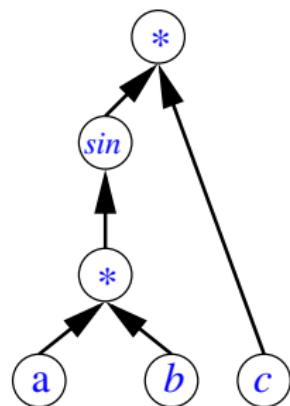
checkpointing

summary

# how does AD compute derivatives?

$$f : y = \sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$$

yields a graph representing the order of computation:

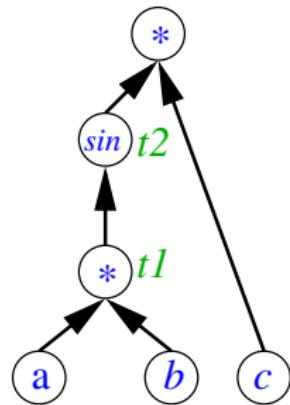


# how does AD compute derivatives?

$$f : y = \sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$$

yields a graph representing the order of computation:

- *code list* → intermediate values  $t1$  and  $t2$



$$t1 = a * b$$

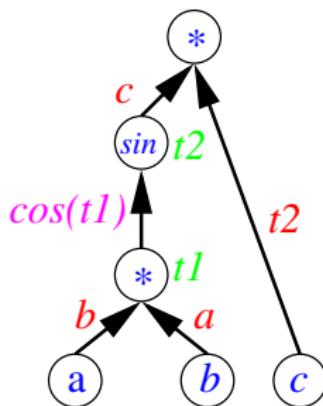
$$t2 = \sin(t1)$$

$$y = t2 * c$$

# how does AD compute derivatives?

$$f : y = \sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$$

yields a graph representing the order of computation:



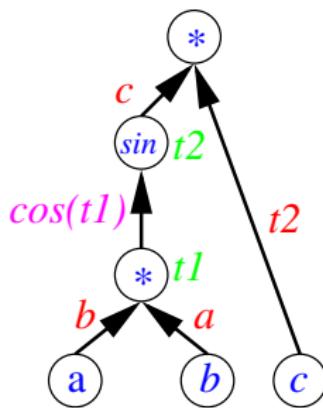
- *code list* → intermediate values  $t1$  and  $t2$
- each intrinsic  $v = \phi(w, u)$  has local partials  $\frac{\partial \phi}{\partial w}, \frac{\partial \phi}{\partial u}$
- e.g.  $\sin(t1)$  yields  $p1 = \cos(t1)$
- in our example all others are already stored in variables

$$\begin{aligned} t1 &= a * b \\ p1 &= \cos(t1) \\ t2 &= \sin(t1) \\ y &= t2 * c \end{aligned}$$

# how does AD compute derivatives?

$$f : y = \sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$$

yields a graph representing the order of computation:



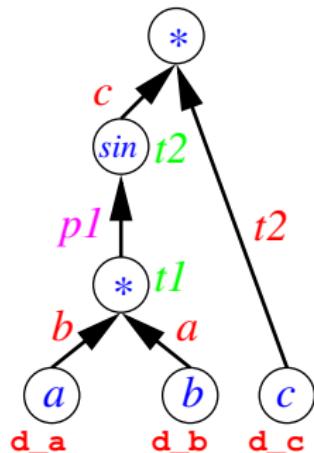
- *code list* → intermediate values  $t1$  and  $t2$
- each intrinsic  $v = \phi(w, u)$  has local partials  $\frac{\partial \phi}{\partial w}, \frac{\partial \phi}{\partial u}$
- e.g.  $\sin(t1)$  yields  $p1 = \cos(t1)$
- in our example all others are already stored in variables

$$\begin{aligned} t1 &= a * b \\ p1 &= \cos(t1) \\ t2 &= \sin(t1) \\ y &= t2 * c \end{aligned}$$

What do we do with this?

# forward mode with directional derivatives

- **associate** each variable  $v$  with a derivative  $\dot{v}$
- take a point  $(a_0, b_0, c_0)$  and a direction  $(\dot{a}, \dot{b}, \dot{c})$
- for each  $v = \phi(w, u)$  propagate forward in order  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$



- in practice: associate *by name* [ $a$ ,  $d_a$ ]  
or *by address* [ $a\%v$ ,  $a\%d$ ]
- interleave propagation computations

$$t_1 = a * b$$

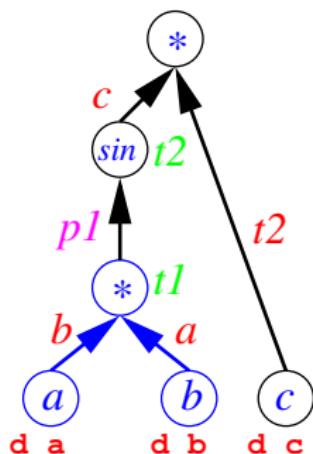
$$p_1 = \cos(t_1)$$

$$t_2 = \sin(t_1)$$

$$y = t_2 * c$$

# forward mode with directional derivatives

- **associate** each variable  $v$  with a derivative  $\dot{v}$
- take a point  $(a_0, b_0, c_0)$  and a direction  $(\dot{a}, \dot{b}, \dot{c})$
- for each  $v = \phi(w, u)$  propagate forward in order  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$



- in practice: associate *by name* [ $a$ ,  $d_a$ ]  
or *by address* [ $a\%v$ ,  $a\%d$ ]
- interleave propagation computations

$$t1 = a * b$$

$$d_{t1} = d_a * b + d_b * a$$

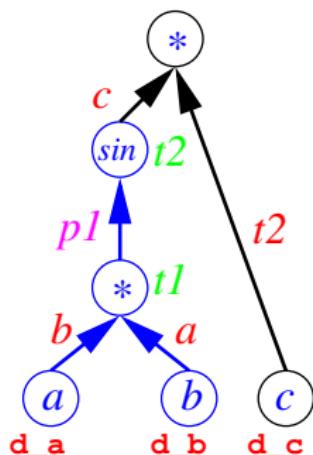
$$p1 = \cos(t1)$$

$$t2 = \sin(t1)$$

$$y = t2 * c$$

# forward mode with directional derivatives

- **associate** each variable  $v$  with a derivative  $\dot{v}$
- take a point  $(a_0, b_0, c_0)$  and a direction  $(\dot{a}, \dot{b}, \dot{c})$
- for each  $v = \phi(w, u)$  propagate forward in order  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$



- in practice: associate *by name* [ $a, d_a$ ] or *by address* [ $a\%v, a\%d$ ]
- interleave propagation computations

$$t1 = a * b$$

$$d_t1 = d_a * b + d_b * a$$

$$p1 = \cos(t1)$$

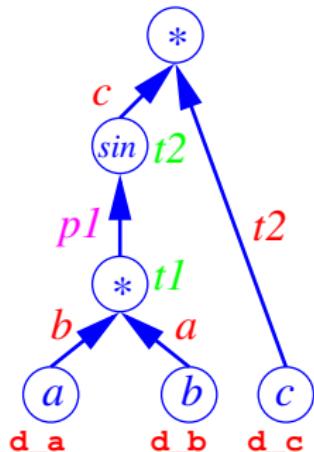
$$t2 = \sin(t1)$$

$$d_t2 = d_t1 * p1$$

$$y = t2 * c$$

# forward mode with directional derivatives

- **associate** each variable  $v$  with a derivative  $\dot{v}$
- take a point  $(a_0, b_0, c_0)$  and a direction  $(\dot{a}, \dot{b}, \dot{c})$
- for each  $v = \phi(w, u)$  propagate forward in order  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$



- in practice: associate *by name* [ $a, d_a$ ] or *by address* [ $a\%v, a\%d$ ]
- interleave propagation computations

$$t1 = a * b$$

$$d_t1 = d_a * b + d_b * a$$

$$p1 = \cos(t1)$$

$$t2 = \sin(t1)$$

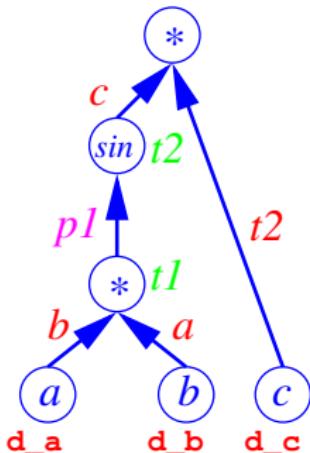
$$d_t2 = d_t1 * p1$$

$$y = t2 * c$$

$$d_y = d_t2 * c + d_c * t2$$

# forward mode with directional derivatives

- **associate** each variable  $v$  with a derivative  $\dot{v}$
- take a point  $(a_0, b_0, c_0)$  and a direction  $(\dot{a}, \dot{b}, \dot{c})$
- for each  $v = \phi(w, u)$  propagate forward in order  $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$



- in practice: associate *by name* [ $a$ ,  $d_a$ ]  
or *by address* [ $a\%v$ ,  $a\%d$ ]
- interleave propagation computations

$$t_1 = a * b$$

$$d_{t_1} = d_a * b + d_b * a$$

$$p_1 = \cos(t_1)$$

$$t_2 = \sin(t_1)$$

$$d_{t_2} = d_{t_1} * p_1$$

$$y = t_2 * c$$

$$d_y = d_{t_2} * c + d_c * t_2$$

What is in  $d_y$  ?

## **d\_y** contains a projection

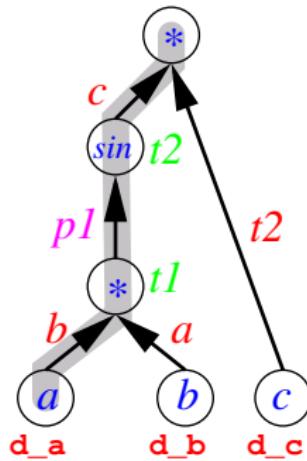
- $\dot{y} = J\dot{x}$  computed at  $x_0$

## **d\_y** contains a projection

- $\dot{y} = J\dot{x}$  computed at  $x_0$
- for example for  $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$

## $\text{d\_y}$ contains a projection

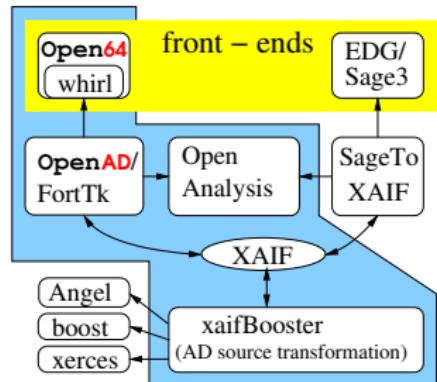
- $\dot{\mathbf{y}} = \mathbf{J}\dot{\mathbf{x}}$  computed at  $\mathbf{x}_0$
- for example for  $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$



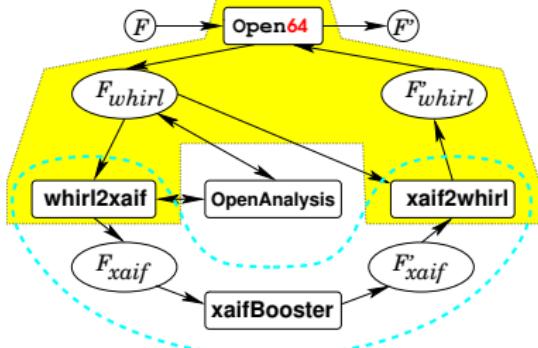
- yields the first element of the gradient
- all gradient elements cost  $\mathcal{O}(n)$  function evaluations

# sidebar: OpenAD overview

- [www.mcs.anl.gov/OpenAD](http://www.mcs.anl.gov/OpenAD)
- forward and **reverse**
- source transformation
- modular design
- aims at large problems
- language independent transformation
- researching combinatorial problems
- current Fortran front-end Open64 (Open64/SL branch at Rice U)
- migration to Rose (already used for C/C++ with EDG)
- uses *association by address* (i.e. has an active type)
- Rapsodia for higher-order derivatives via type change transformation



Fortran pipeline:



# sidebar: toy example

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y

  y=sin(x*x)

end subroutine
```

# sidebar: toy example

## prepare original code

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

## sidebar: toy example

prepare original code  $\Rightarrow$  run it through OpenAD

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oadS_0
...
REAL(w2f__8) oadS_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_5 = ((oadS_3 + oadS_2) * oadS_1)
CALL sax(oadS_5,X,Y)
RETURN
END SUBROUTINE
```

## sidebar: toy example

prepare original code  $\Rightarrow$  run it through OpenAD

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oadS_0
...
REAL(w2f__8) oadS_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_5 = ((oadS_3 + oadS_2) * oadS_1)
CALL sax(oadS_5,X,Y)
RETURN
END SUBROUTINE
```

## sidebar: toy example

prepare original code  $\Rightarrow$  run it through OpenAD

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oads_0
...
REAL(w2f__8) oads_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
oads_0 = (X%v*X%v)
Y%v = SIN(oads_0)
oads_2 = X%v
oads_3 = X%v
oads_1 = COS(oads_0)
oads_5 = ((oads_3 + oads_2) * oads_1)
CALL sax(oads_5,X,Y)
RETURN
END SUBROUTINE
```

## sidebar: toy example

prepare original code  $\Rightarrow$  run it through OpenAD  $\Rightarrow$  adapt a driver routine

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver:

```
program driver
  implicit none
  external head
  real::: x, y
  x=.5D0
  call head(x,y)
end program driver
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oads_0
...
REAL(w2f__8) oads_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
oads_0 = (X%v*X%v)
Y%v = SIN(oads_0)
oads_2 = X%v
oads_3 = X%v
oads_1 = COS(oads_0)
oads_5 = ((oads_3 + oads_2) * oads_1)
CALL sax(oads_5,X,Y)
RETURN
END SUBROUTINE
```

## sidebar: toy example

prepare original code  $\Rightarrow$  run it through OpenAD  $\Rightarrow$  adapt a driver routine

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver:

```
program driver
  use OAD_active
  implicit none
  external head
  type(active):: x, y
  x%v=.5D0
  x%d=1.0
  call head(x,y)
  print *, "F(1,1)=" ,y%d
end program driver
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oads_0
...
REAL(w2f__8) oads_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
oads_0 = (X%v*X%v)
Y%v = SIN(oads_0)
oads_2 = X%v
oads_3 = X%v
oads_1 = COS(oads_0)
oads_5 = ((oads_3 + oads_2) * oads_1)
CALL sax(oads_5,X,Y)
RETURN
END SUBROUTINE
```

## sidebar: toy example

prepare original code  $\Rightarrow$  run it through OpenAD  $\Rightarrow$  adapt a driver routine

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver:

```
program driver
  use OAD_active
  implicit none
  external head
  type(active):: x, y
  x%v=.5D0
  x%d=1.0
  call head(x,y)
  print *, "F(1,1)=" ,y%d
end program driver
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oads_0
...
REAL(w2f__8) oads_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
oads_0 = (X%v*X%v)
Y%v = SIN(oads_0)
oads_2 = X%v
oads_3 = X%v
oads_1 = COS(oads_0)
oads_5 = ((oads_3 + oads_2) * oads_1)
CALL sax(oads_5,X,Y)
RETURN
END SUBROUTINE
```

the `sax` call comes from propagation following *preaccumulation*...not discussed yet

# applications

for instance

- ocean/atmosphere state estimation & uncertainty quantification, oil reservoir modeling
- computational chemical engineering
- airfoil shape optimization, suspended droplets, ...
- beam physics
- mechanical engineering (design optimization)

use

- **gradients**
- Jacobian projections
- Hessian projections
- higher order derivatives  
(full or partial tensors, univariate Taylor series)

# applications

for instance

- ocean/atmosphere state estimation & uncertainty quantification, oil reservoir modeling
- computational chemical engineering
- airfoil shape optimization, suspended droplets, ...
- beam physics
- mechanical engineering (design optimization)

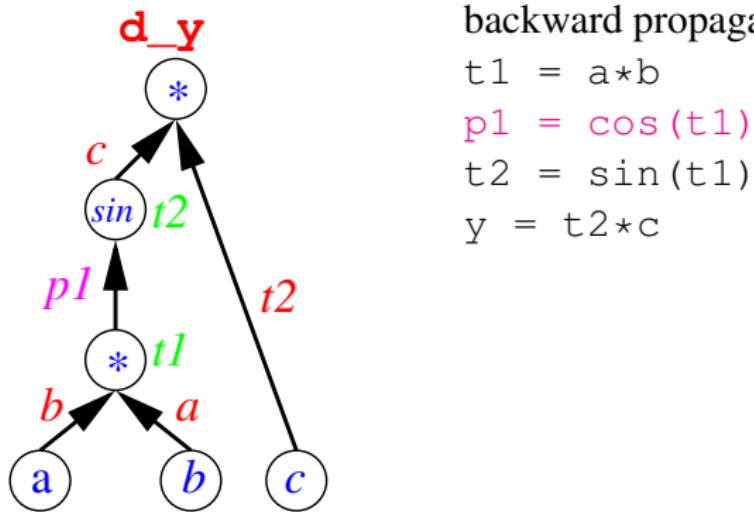
use

- **gradients**
- Jacobian projections
- Hessian projections
- higher order derivatives  
(full or partial tensors, univariate Taylor series)

How do we get the cheap gradients?

## reverse mode with adjoints

- same association model
- take a point  $(a_0, b_0, c_0)$ , compute  $y$ , pick a weight  $\bar{y}$
- for each  $v = \phi(w, u)$  propagate backward  
 $\bar{w}+ = \frac{\partial\phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial\phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

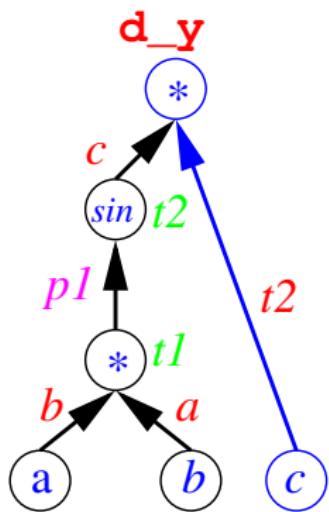


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
```

## reverse mode with adjoints

- same association model
- take a point  $(a_0, b_0, c_0)$ , compute  $y$ , pick a weight  $\bar{y}$
- for each  $v = \phi(w, u)$  propagate backward  
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

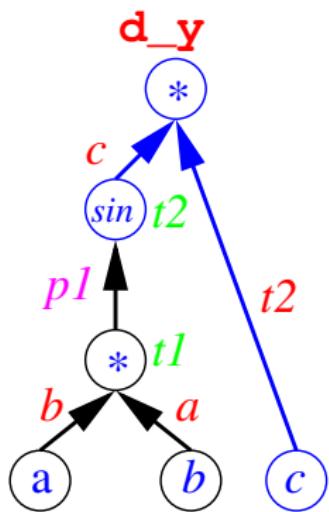


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
```

## reverse mode with adjoints

- same association model
- take a point  $(a_0, b_0, c_0)$ , compute  $y$ , pick a weight  $\bar{y}$
- for each  $v = \phi(w, u)$  propagate backward  
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

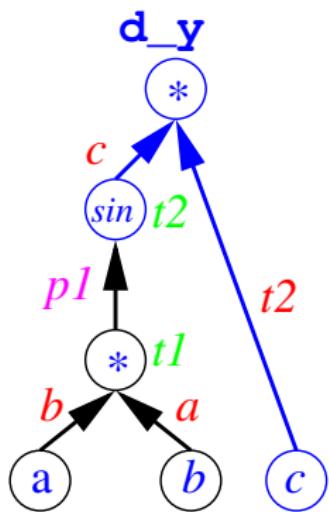


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
```

## reverse mode with adjoints

- same association model
- take a point  $(a_0, b_0, c_0)$ , compute  $y$ , pick a weight  $\bar{y}$
- for each  $v = \phi(w, u)$  propagate backward  
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

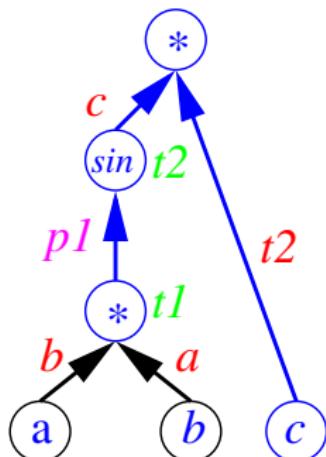


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
```

## reverse mode with adjoints

- same association model
- take a point  $(a_0, b_0, c_0)$ , compute  $y$ , pick a weight  $\bar{y}$
- for each  $v = \phi(w, u)$  propagate backward  
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

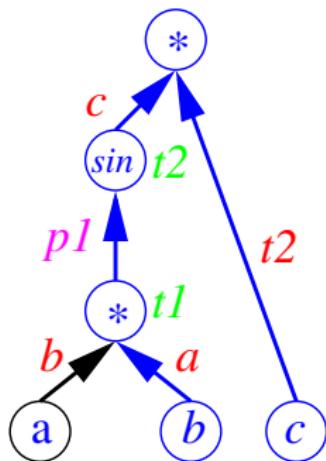


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
```

## reverse mode with adjoints

- same association model
- take a point  $(a_0, b_0, c_0)$ , compute  $y$ , pick a weight  $\bar{y}$
- for each  $v = \phi(w, u)$  propagate backward  
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

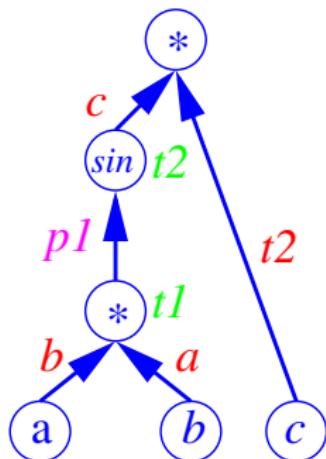


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
```

## reverse mode with adjoints

- same association model
- take a point  $(a_0, b_0, c_0)$ , compute  $y$ , pick a weight  $\bar{y}$
- for each  $v = \phi(w, u)$  propagate backward  
 $\bar{w}+ = \frac{\partial\phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial\phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

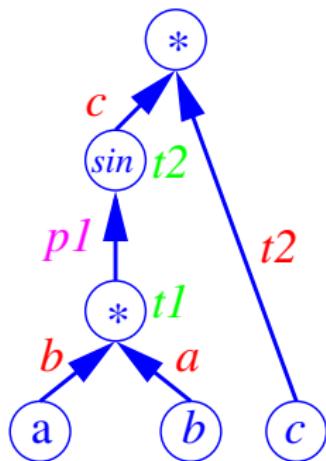


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
d_a = b*d_t1
```

## reverse mode with adjoints

- same association model
- take a point  $(a_0, b_0, c_0)$ , compute  $y$ , pick a weight  $\bar{y}$
- for each  $v = \phi(w, u)$  propagate backward  
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$



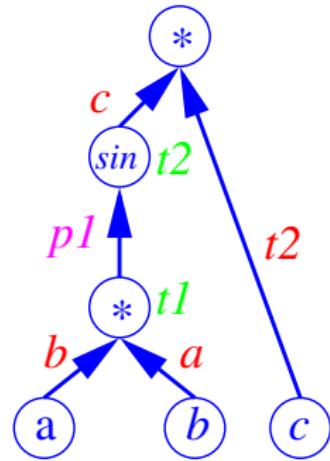
backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
d_a = b*d_t1
```

What is in  $(d_a, d_b, d_c)$ ?

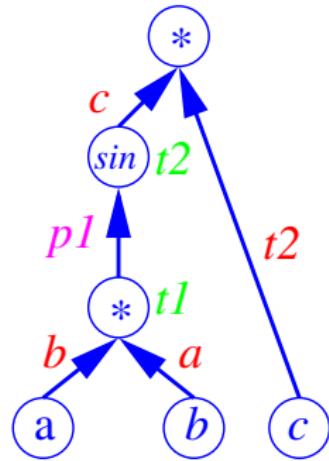
$(d\_a, d\_b, d\_c)$  contains a projection

- $\bar{x} = \bar{y}^T J$  computed at  $x_0$



$(d_a, d_b, d_c)$  contains a projection

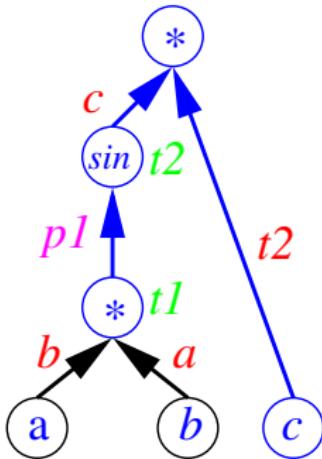
- $\bar{x} = \bar{y}^T J$  computed at  $x_0$
- for example for  $\bar{y} = 1$  we have  $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost  $\mathcal{O}(1)$  function evaluations

## $(d_a, d_b, d_c)$ contains a projection

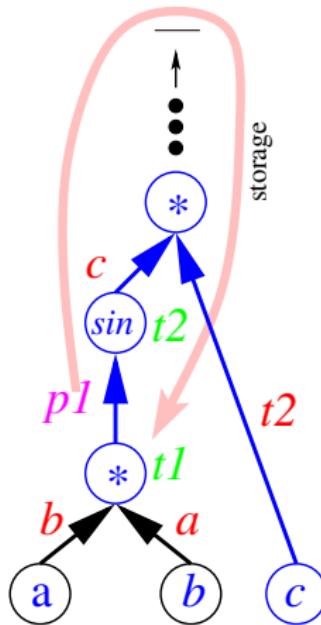
- $\bar{x} = \bar{y}^T J$  computed at  $x_0$
- for example for  $\bar{y} = 1$  we have  $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost  $\mathcal{O}(1)$  function evaluations
- but consider when  $p1$  is computed and when it is used

## $(d_a, d_b, d_c)$ contains a projection

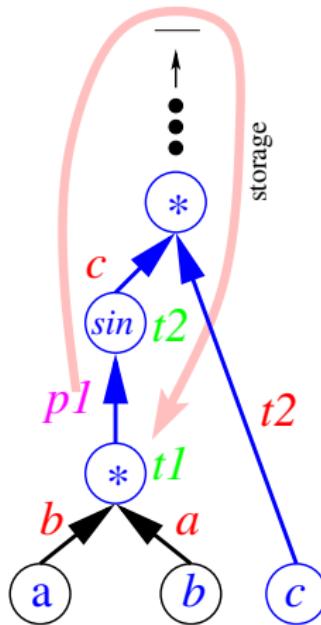
- $\bar{x} = \bar{y}^T J$  computed at  $x_0$
- for example for  $\bar{y} = 1$  we have  $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost  $\mathcal{O}(1)$  function evaluations
- but consider when  $p1$  is computed and when it is used
- **storage requirements** grow with the length of the computation
- typically mitigated by recomputation from checkpoints

## $(d_a, d_b, d_c)$ contains a projection

- $\bar{x} = \bar{y}^T J$  computed at  $x_0$
- for example for  $\bar{y} = 1$  we have  $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$

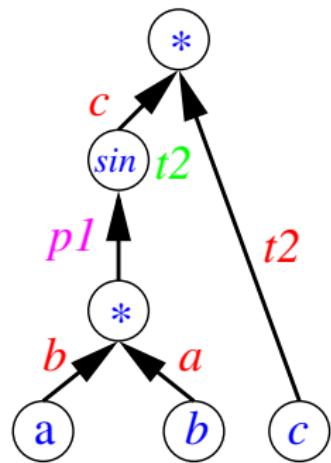


- all gradient elements cost  $\mathcal{O}(1)$  function evaluations
- but consider when  $p1$  is computed and when it is used
- **storage requirements** grow with the length of the computation
- typically mitigated by recomputation from checkpoints

Reverse mode as a source transformation with OpenAD.

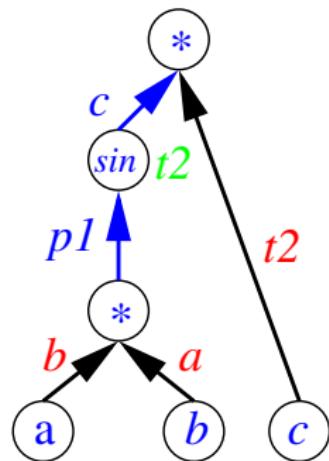
## sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians  $J$
- long program with control flow  $\Rightarrow$  sequence of graphs  $\Rightarrow$  sequence of  $J_i$



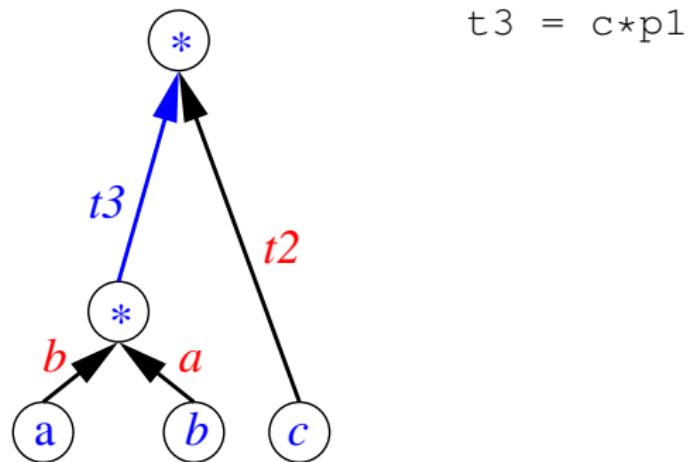
## sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians  $J$
- long program with control flow  $\Rightarrow$  sequence of graphs  $\Rightarrow$  sequence of  $J_i$



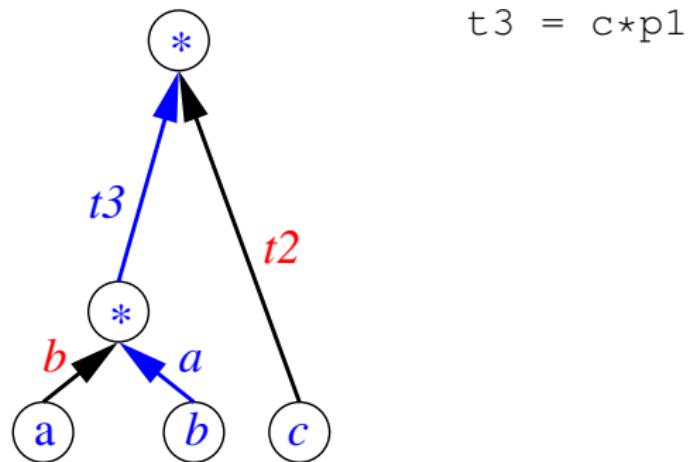
## sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians  $J$
- long program with control flow  $\Rightarrow$  sequence of graphs  $\Rightarrow$  sequence of  $J_i$



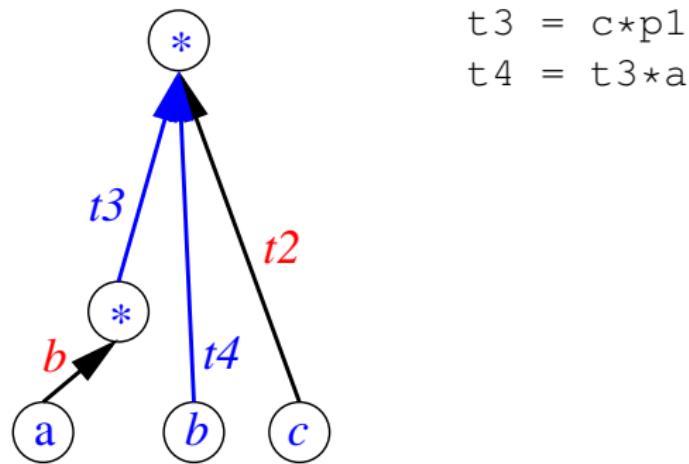
## sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians  $J$
- long program with control flow  $\Rightarrow$  sequence of graphs  $\Rightarrow$  sequence of  $J_i$



## sidebar: preaccumulation & propagation

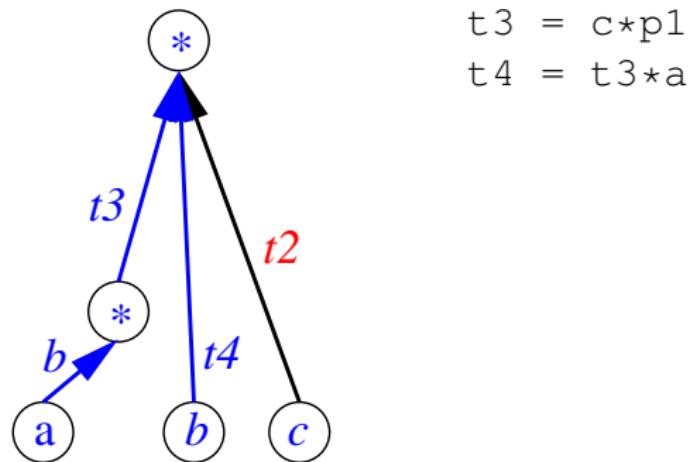
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians  $J$
- long program with control flow  $\Rightarrow$  sequence of graphs  $\Rightarrow$  sequence of  $J_i$



$$\begin{aligned} t3 &= c * p1 \\ t4 &= t3 * a \end{aligned}$$

## sidebar: preaccumulation & propagation

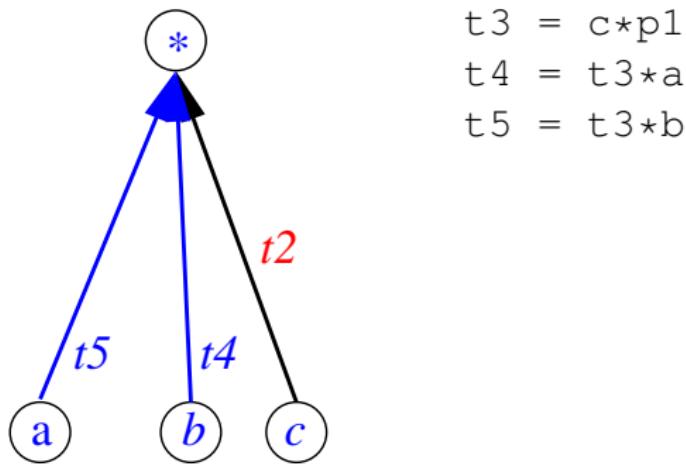
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians  $J$
- long program with control flow  $\Rightarrow$  sequence of graphs  $\Rightarrow$  sequence of  $J_i$



$$\begin{aligned} t3 &= c * p1 \\ t4 &= t3 * a \end{aligned}$$

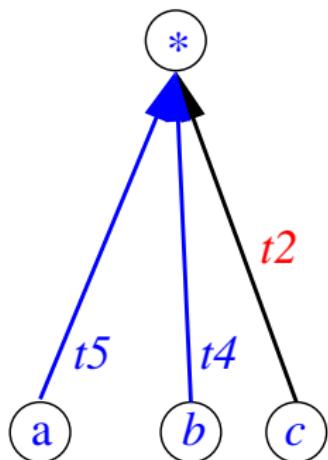
## sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians  $J$
- long program with control flow  $\Rightarrow$  sequence of graphs  $\Rightarrow$  sequence of  $J_i$


$$\begin{aligned} t3 &= c * p1 \\ t4 &= t3 * a \\ t5 &= t3 * b \end{aligned}$$

## sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians  $J$
- long program with control flow  $\Rightarrow$  sequence of graphs  $\Rightarrow$  sequence of  $J_i$



$t_3 = c * p_1$   
 $t_4 = t_3 * a$   
 $t_5 = t_3 * b$

- $(t_5, t_4, t_2)$  is the preaccumulated  $J_i$
- $\min_{ops}(\text{preaccumulation})$  ?  
is a combinatorial problem  
 $\Rightarrow$  compile time AD optimization!
- forward propagation of  $\dot{x}$   
 $(J_k \circ \dots \circ (J_1 \circ \dot{x}) \dots)$
- adjoint propagation of  $\bar{y}$   
 $(\dots (\bar{y}^T \circ J_k) \circ \dots \circ J_1)$

# sidebar: toy example - reverse mode

same code preparation

numerical “model” program:

```
subroutine head(x,y)
    double precision,intent(in) :: x
    double precision,intent(out) :: y
!$openad INDEPENDENT(x)
    y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

## sidebar: toy example - reverse mode

same code preparation  $\Rightarrow$  reverse mode OpenAD pipeline

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store  $J_i$ :

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

## sidebar: toy example - reverse mode

same code preparation  $\Rightarrow$  reverse mode OpenAD pipeline

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store  $J_i$ :

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

## sidebar: toy example - reverse mode

same code preparation  $\Rightarrow$  reverse mode OpenAD pipeline

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store  $J_i$ :

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

# sidebar: toy example - reverse mode

same code preparation  $\Rightarrow$  reverse mode OpenAD pipeline

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store  $J_i$ :

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored  $J_i$  & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

# sidebar: toy example - reverse mode

same code preparation  $\Rightarrow$  reverse mode OpenAD pipeline

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store  $J_i$ :

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored  $J_i$  & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

# sidebar: toy example - reverse mode

same code preparation  $\Rightarrow$  reverse mode OpenAD pipeline

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store  $J_i$ :

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored  $J_i$  & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

# sidebar: toy example - reverse mode

same code preparation  $\Rightarrow$  reverse mode OpenAD pipeline  
 $\Rightarrow$  adapt the driver routine

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver modified for reverse mode:

```
program driver
use OAD_active
implicit none
external head
type(active):: x, y
x%v=.5D0
y%d=1.0
our_rev_mode%tape=.TRUE.
call head(x,y)
print *, "F(1,1)=",x%d
end program driver
```

preaccumulation & store  $J_i$ :

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored  $J_i$  & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

# sidebar: toy example - reverse mode

same code preparation  $\Rightarrow$  reverse mode OpenAD pipeline  
 $\Rightarrow$  adapt the driver routine

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver modified for reverse mode:

```
program driver
use OAD_active
implicit none
external head
type(active):: x, y
x%v=.5D0
y%d=1.0
our_rev_mode%tape=.TRUE.
call head(x,y)
print *, "F(1,1)=", x%d
end program driver
```

preaccumulation & store  $J_i$ :

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored  $J_i$  & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

# outline

motivation

basics & examples

- simple forward

- forward with OpenAD

- sample applications

- simple reverse

- preaccumulation & propagation

- reverse with OpenAD

tools and user concerns

- tools

- match tool and application

- source transformation vs. operator overloading

- forward vs. reverse

- checkpointing

summary

## some tools

source transformation tools have varying levels of:

- language coverage
- flexibility
- functionality
- availability

a few select examples:

- open source and free: OpenAD
- open source with registration and binary-only front-end: ADIC
- binary only, free for academic users: Adifor, Tapenade
- commercial: TAF (FastOpt), nagf95 (NAG; under development)
- ADiMat
- various defunct projects...
- comprehensive AD tool collection at [www.autodiff.org](http://www.autodiff.org)

# pick the tool based on the application

match application requirements with AD techniques

- knowing AD tool “internal” algorithms is of interest to the user  
(e.g. compare to compiler vectorization or interval arithmetic)
- simple models with low computational complexity  
→ can get away with “something”
- fully automatic solutions exist for narrowly defined setups (e.g. NEOS)
- complicated models → tool applicability?
- high computational complexity → efficiency of derivative computations ?
- tool availability (e.g. source transformation for C++ ?)

# source transformation vs. operator overloading

- complicated implementation of tools
- especially for reverse mode
- full front end, back end, analysis
- efficiency gains from
  - **compile time AD optimizations**
  - activity analysis
  - explicit control flow reversal
- source transformation based type change & overloaded operators appropriate for higher-order derivatives.
- efficiency depends on analysis accuracy
- simple tool implementation
- reverse mode: generate & reinterpret an execution trace → inefficient
- implemented as a library
- efficiency gains from:
  - runtime AD optimization
  - optimized library
  - inlining (for low order)
- manual type change
- ↛ formatted I/O, allocation,...
- matching signatures (Fortran)
- easier with templates

---

higher-order derivatives  $\Rightarrow$  source transformation based type change  
+ overloaded operators.

## forward vs. reverse

- simplest rule: given  $y = f(x) : \mathbb{R}^n \mapsto \mathbb{R}^m$  use reverse if  $n \gg m$  (gradient)
- what if  $n \approx m$  and large
  - want only projections, e.g.  $J\dot{x}$
  - sparsity (e.g. of the Jacobian)
  - partial separability (e.g.  $f(x) = \sum(f_i(x_i)), x_i \in \mathcal{D}_i \Subset \mathcal{D} \ni x$ )
  - intermediate interfaces of different size
- the above may make forward mode feasible  
(projection  $\bar{y}^T J$  requires reverse)
- higher order tensors (practically feasible for small  $n$ ) → forward mode  
(reverse mode saves factor  $n$  in effort only once)
- this determines overall propagation direction, not necessarily the local preaccumulation (combinatorial problem)

# use of checkpointing to mitigate storage requirements



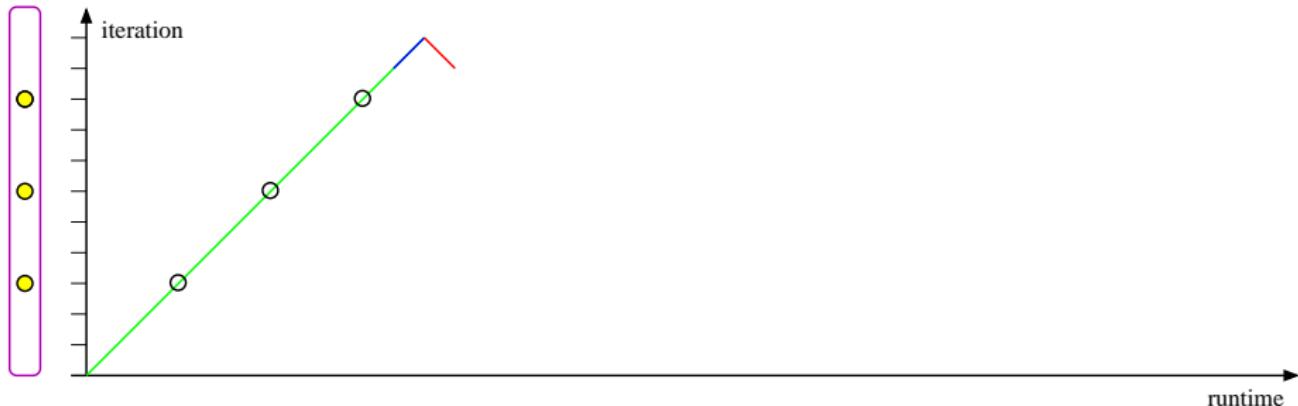
- 11 iters.

# use of checkpointing to mitigate storage requirements



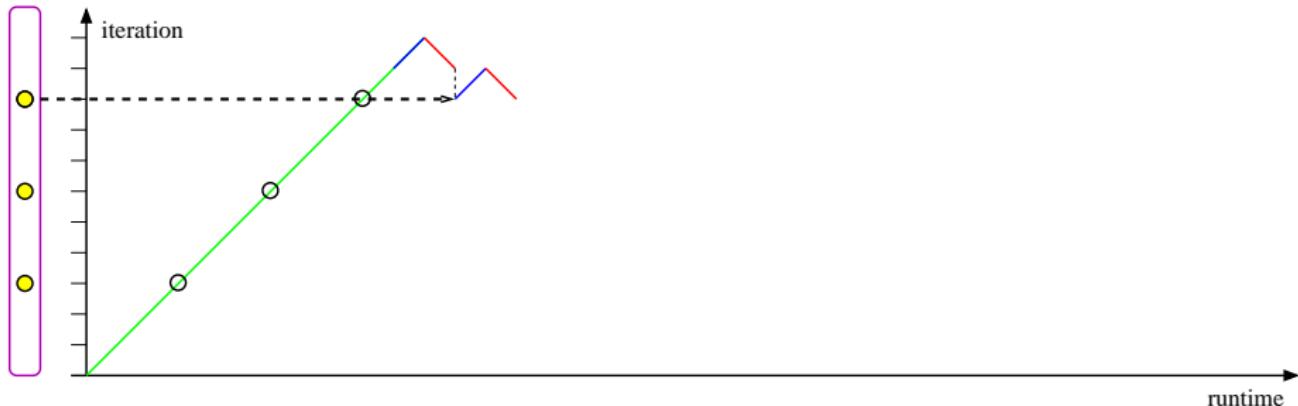
- 11 iters., memory limited to one iter. of storing  $J_i$
- run forward, **store** the last step, and **adjoin**

# use of checkpointing to mitigate storage requirements



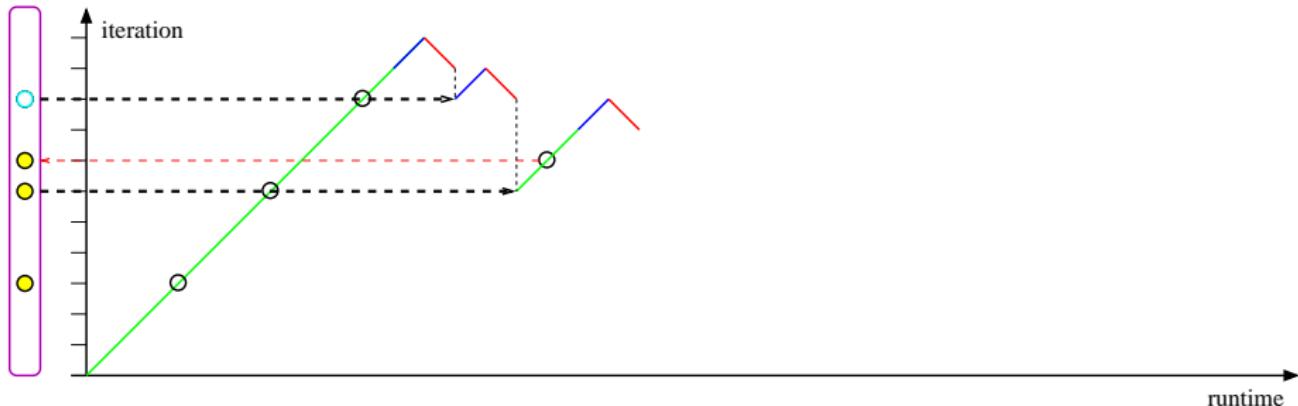
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin

# use of checkpointing to mitigate storage requirements



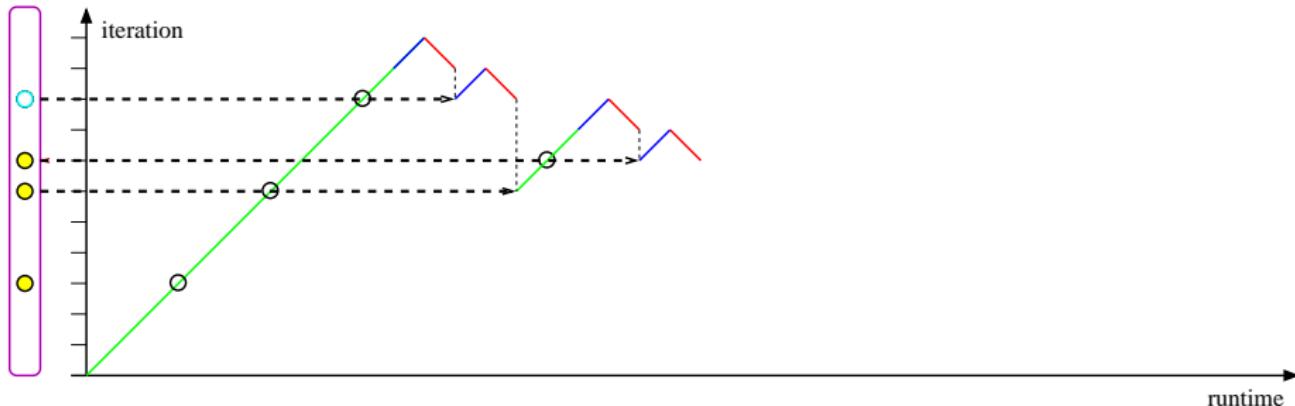
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute

## use of checkpointing to mitigate storage requirements



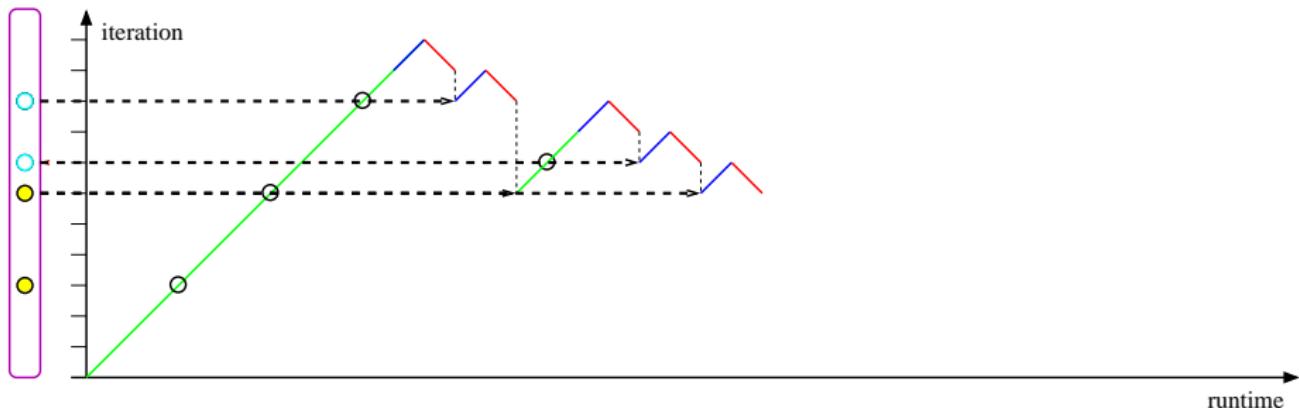
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

## use of checkpointing to mitigate storage requirements



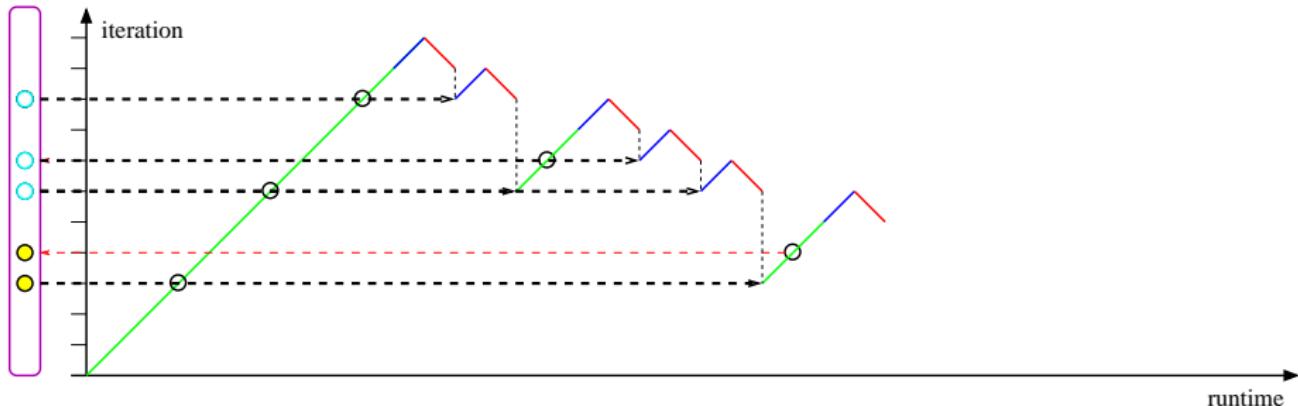
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, **store** the last step, and **adjoin**
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

# use of checkpointing to mitigate storage requirements



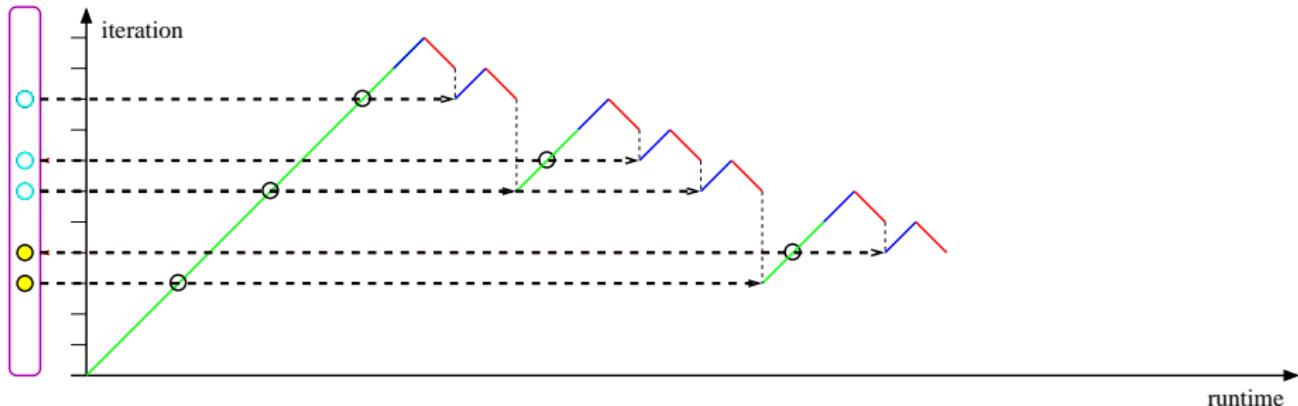
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

## use of checkpointing to mitigate storage requirements



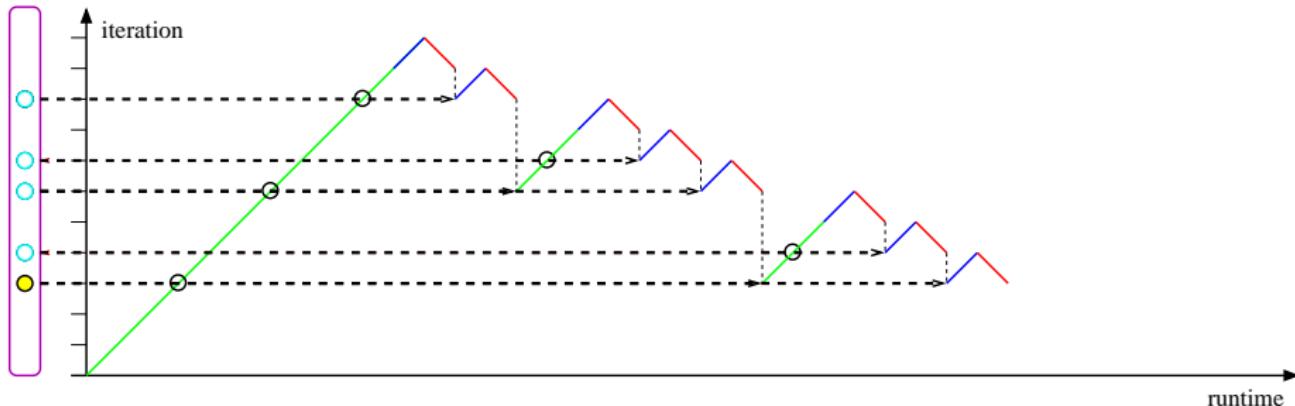
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, **store** the last step, and **adjoin**
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

## use of checkpointing to mitigate storage requirements



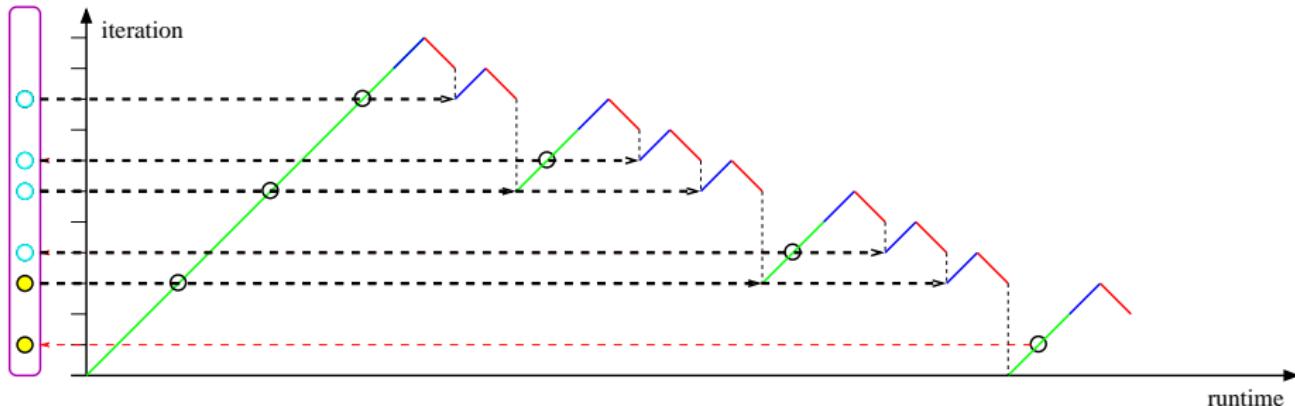
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

# use of checkpointing to mitigate storage requirements



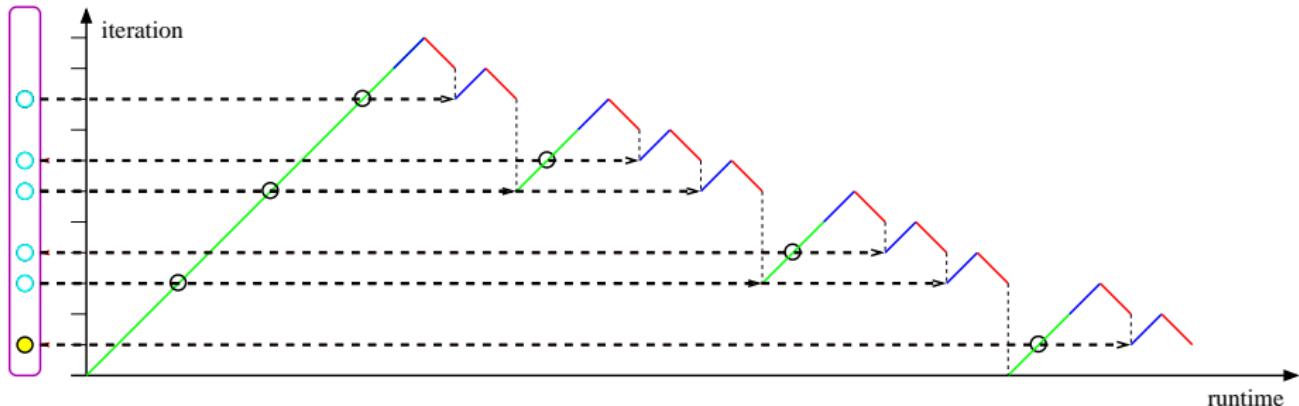
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

# use of checkpointing to mitigate storage requirements



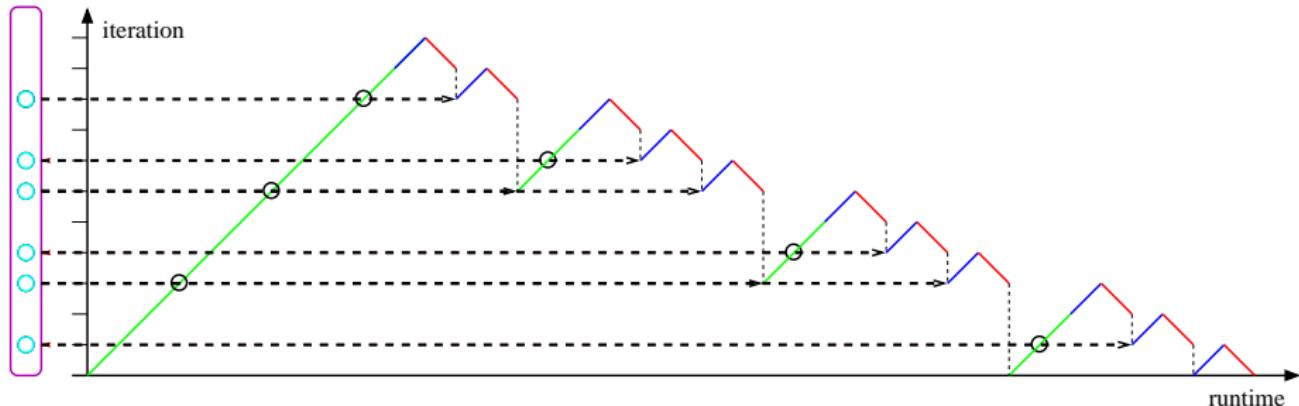
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, **store** the last step, and **adjoin**
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

# use of checkpointing to mitigate storage requirements



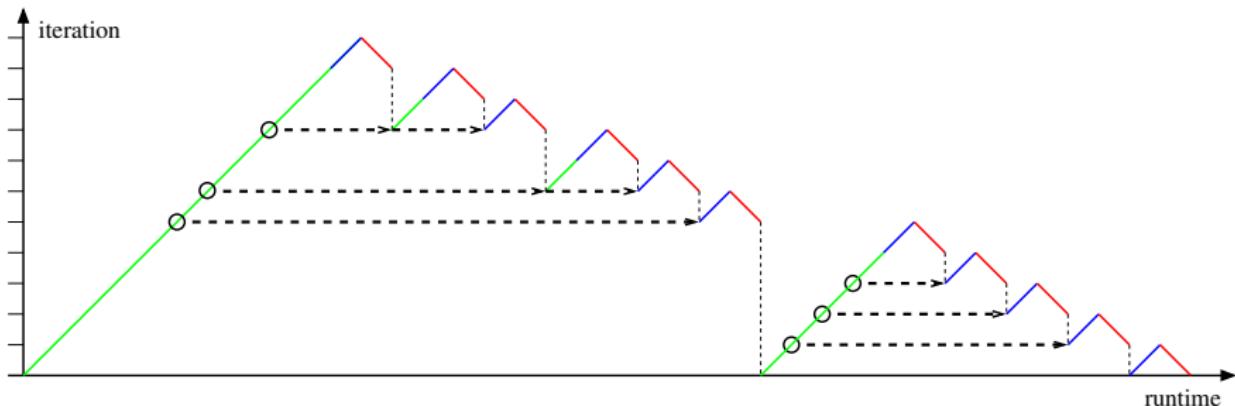
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

## use of checkpointing to mitigate storage requirements



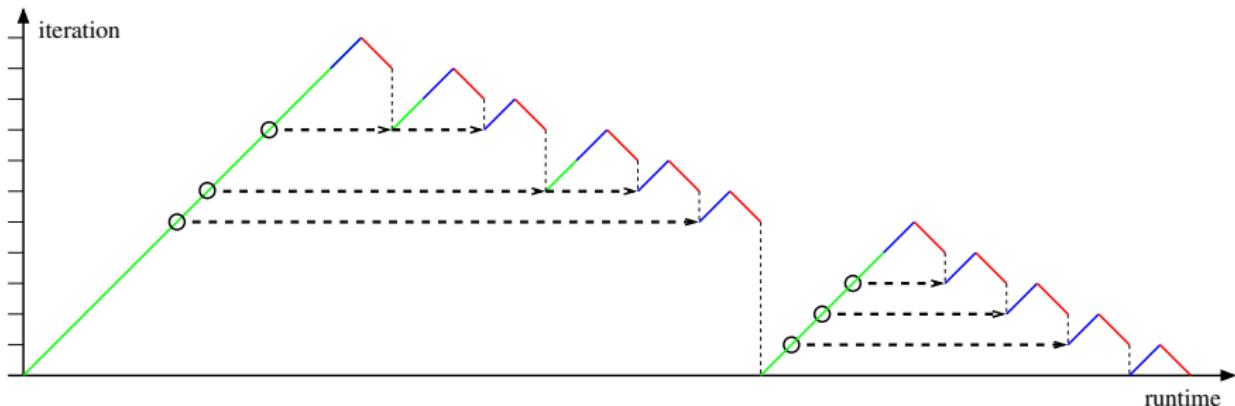
- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

# use of checkpointing to mitigate storage requirements



- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
- optimal (binomial) scheme encoded in revolve; F9X implementation available at <http://mercurial.mcs.anl.gov/ad/RevolveF9X>

# use of checkpointing to mitigate storage requirements



- 11 iters., memory limited to one iter. of storing  $J_i$  & 3 checkpoints
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
- optimal (binomial) scheme encoded in revolve; F9X implementation available at <http://mercurial.mcs.anl.gov/ad/RevolveF9X>
- source transformation tool needs to provide four variants

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
  case (rvStore)
    call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
    call cp_close()
  case (rvRestore)
    call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
    currIter=rvAct%iteration
    call cp_close()
  case (rvForward)
    oadRevMod%plain=.TRUE.
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.FALSE.
    do while (currIter<rvAct%iteration)
      call loopBody(x)
      currIter=currIter+1
    end do
  case (rvFirstUTurn)
    oadRevMod%plain=.FALSE.
    oadRevMod%tape=.TRUE.
    oadRevMod%adjoint=.FALSE.
    call loopBody(x)
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.TRUE.
    call loopBody(x)
  case (rvUTurn)...
  end select
end do
```

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

## • init revolve

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
  case (rvStore)
    call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
    call cp_close()
  case (rvRestore)
    call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
    currIter=rvAct%iteration
    call cp_close()
  case (rvForward)
    oadRevMod%plain=.TRUE.
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.FALSE.
    do while (currIter<rvAct%iteration)
      call loopBody(x)
      currIter=currIter+1
    end do
  case (rvFirstUTurn)
    oadRevMod%plain=.FALSE.
    oadRevMod%tape=.TRUE.
    oadRevMod%adjoint=.FALSE.
    call loopBody(x)
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.TRUE.
    call loopBody(x)
  case (rvUTurn)...
  end select
end do
```

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init revolve
- revolve loop

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
  case (rvStore)
    call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
    call cp_close()
  case (rvRestore)
    call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
    currIter=rvAct%iteration
    call cp_close()
  case (rvForward)
    oadRevMod%plain=.TRUE.
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.FALSE.
    do while (currIter<rvAct%iteration)
      call loopBody(x)
      currIter=currIter+1
    end do
  case (rvFirstUTurn)
    oadRevMod%plain=.FALSE.
    oadRevMod%tape=.TRUE.
    oadRevMod%adjoint=.FALSE.
    call loopBody(x)
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.TRUE.
    call loopBody(x)
  case (rvUTurn)...
  end select
end do
```

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init revolve
- revolve loop
- get the action

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
  case (rvStore)
    call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
    call cp_close()
  case (rvRestore)
    call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
    currIter=rvAct%iteration
    call cp_close()
  case (rvForward)
    oadRevMod%plain=.TRUE.
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.FALSE.
    do while (currIter<rvAct%iteration)
      call loopBody(x)
      currIter=currIter+1
    end do
  case (rvFirstUTurn)
    oadRevMod%plain=.FALSE.
    oadRevMod%tape=.TRUE.
    oadRevMod%adjoint=.FALSE.
    call loopBody(x)
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.TRUE.
    call loopBody(x)
  case (rvUTurn)...
  end select
end do
```

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init revolve
- revolve loop
- get the action
- transformation provides:
  - store checkpoint

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
    case (rvStore)
      call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
      call cp_close()
    case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
    case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
    case (rvFirstUTurn)
      oadRevMod%plain=.FALSE.
      oadRevMod%tape=.TRUE.
      oadRevMod%adjoint=.FALSE.
      call loopBody(x)
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.TRUE.
      call loopBody(x)
    case (rvUTurn)...
    end select
  end do
```

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init revolve
- revolve loop
- get the action
- transformation provides:
  - store checkpoint
  - restore checkpoint

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
  case (rvStore)
    call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
    call cp_close()
    case (rvRestore)
      call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
      currIter=rvAct%iteration
      call cp_close()
    case (rvForward)
      oadRevMod%plain=.TRUE.
      oadRevMod%tape=.FALSE.
      oadRevMod%adjoint=.FALSE.
      do while (currIter<rvAct%iteration)
        call loopBody(x)
        currIter=currIter+1
      end do
      case (rvFirstUTurn)
        oadRevMod%plain=.FALSE.
        oadRevMod%tape=.TRUE.
        oadRevMod%adjoint=.FALSE.
        call loopBody(x)
        oadRevMod%tape=.FALSE.
        oadRevMod%adjoint=.TRUE.
        call loopBody(x)
      case (rvUTurn)...
      end select
    end do
```

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init revolve
- revolve loop
- get the action
- transformation provides:
  - store checkpoint
  - restore checkpoint
  - forward to a iteration

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
  case (rvStore)
    call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
    call cp_close()
  case (rvRestore)
    call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
    currIter=rvAct%iteration
    call cp_close()
  case (rvForward)
    oadRevMod%plain=.TRUE.
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.FALSE.
    do while (currIter<rvAct%iteration)
      call loopBody(x)
      currIter=currIter+1
    end do
  case (rvFirstUTurn)
    oadRevMod%plain=.FALSE.
    oadRevMod%tape=.TRUE.
    oadRevMod%adjoint=.FALSE.
    call loopBody(x)
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.TRUE.
    call loopBody(x)
  case (rvUTurn)...
  end select
end do
```

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init revolve
- revolve loop
- get the action
- transformation provides:
  - store checkpoint
  - restore checkpoint
  - forward to a iteration
  - **store & adjoint**

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
  case (rvStore)
    call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
    call cp_close()
  case (rvRestore)
    call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
    currIter=rvAct%iteration
    call cp_close()
  case (rvForward)
    oadRevMod%plain=.TRUE.
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.FALSE.
    do while (currIter<rvAct%iteration)
      call loopBody(x)
      currIter=currIter+1
    end do
  case (rvFirstUTurn)
    oadRevMod%plain=.FALSE.
    oadRevMod%tape=.TRUE.
    oadRevMod%adjoint=.FALSE.
    call loopBody(x)
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.TRUE.
    call loopBody(x)
  case (rvUTurn)...
  end select
end do
```

# OpenAD: revolve with a prefab subroutine template

iteration loop:

```
!$openad XXX Template ad_revolve.f
subroutine loopWrapper(x,n)
  double precision :: x
  integer :: n
!$openad INDEPENDENT(x)
  do i=1,n
    call loopBody(x)
  end do
!$openad DEPENDENT(x)
end subroutine
```

- init revolve
- revolve loop
- get the action
- transformation provides:
  - store checkpoint
  - restore checkpoint
  - forward to a iteration
  - **store & adjoint**
- OpenAD has documented examples

```
rc=rvInit(n,cpCnt,errMsg)
do while (rvAct%actionFlag/=rvDone)
  rvAct=rvNextAction()
  select case (rvAct%actionFlag)
  case (rvStore)
    call cp_write_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=4
    call cp_close()
  case (rvRestore)
    call cp_read_open(rvAct%iteration)
!$PLACEHOLDER_PRAGMA$ id=6
    currIter=rvAct%iteration
    call cp_close()
  case (rvForward)
    oadRevMod%plain=.TRUE.
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.FALSE.
    do while (currIter<rvAct%iteration)
      call loopBody(x)
      currIter=currIter+1
    end do
  case (rvFirstUTurn)
    oadRevMod%plain=.FALSE.
    oadRevMod%tape=.TRUE.
    oadRevMod%adjoint=.FALSE.
    call loopBody(x)
    oadRevMod%tape=.FALSE.
    oadRevMod%adjoint=.TRUE.
    call loopBody(x)
  case (rvUTurn)...
  end select
end do
```

# outline

motivation

basics & examples

- simple forward

- forward with OpenAD

- sample applications

- simple reverse

- preaccumulation & propagation

- reverse with OpenAD

tools and user concerns

- tools

- match tool and application

- source transformation vs. operator overloading

- forward vs. reverse

- checkpointing

summary

# summary

things covered: unless I ran out of time

- forward & reverse mode with source transformation
- preaccumulation & checkpointing
- some model properties affecting AD efficiency
- some source transformation tools & OpenAD examples

things not discussed:

- good practices for applying AD
- non-smooth model behavior
- dealing with parallelized models
- using libraries
- handling certain programming language features
- combinatorial problems affecting efficiency

more info and links at:

[www.autodiff.org](http://www.autodiff.org)